RBM Music Generation

Restricted Boltzman Machine

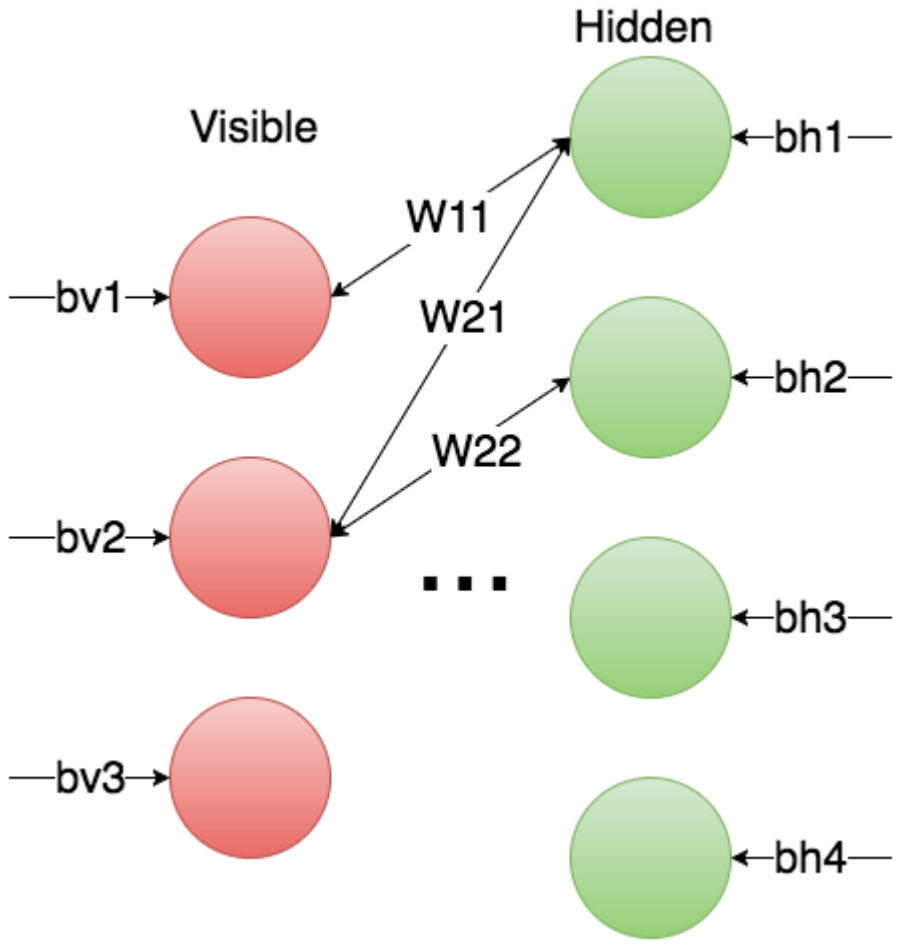
The RBM is a neural network with 2 layers, the visible layer and the hidden layer. Each visible node is connected to each hidden node (and vice versa), but there are no visible-visible or hidden-hidden connections (the RBM is a complete bipartite graph). Since there are only 2 layers, we can fully describe a trained RBM with 3 parameters:

- The weight matrix W:
  - W has size $n_{visible}$ x n_{hidden}. W_{ij} is the weight of the connection between visible node i and hidden node j.

- The bias vector bv:
  - bv is a vector with $n_{visible}$ elements. Element i is the bias for the i^{th} visible node.

- The bias vector bh:
  - bh is a vector n_hidden element. Element j is the bias for thr j^{th} hidden node.

$n_{visible}$ is the number of features in the input vectors. n_{hidden} is the size of the hidden layes.



Sampling

RBMs are generative models that directly model the probability distribution of data and can be used for data augmentation and reconstruction. To sample from an RBM, we perform an algorithm known as Gibbs sampling. Essentially, this algorithm works like this:

- Initialize the visible nodes. We can initialize them randomly, or we can set them equal to an input example.
- Repeat the following process for k steps, or until convergence:

1. Propagate the values of the visible nodes forward, and then sample the new values of the hidden nodes.

    - That is, randomly set the values of each h_i to be 1 with probability $\sigma(W^T v + bh)_i$.

2. Propagate the values of the hidden nodes back to the visible nodes, and sample the new values of the visible nodes.

    - That is, randomly set the values of each $vivi$ to be 1 with probability $\sigma(Wh+bv)_i$

At the end of the algorithm, the visible nodes will store the value of the sample.

Training

When we train an RBM, our goal is to find the values for its parameters that maximize the likelihood of our data being drawn from that distribution.

To do that, we use a very simple strategy:

- Initialize the visible nodes with some vector xx from our dataset
- Sample x(bar) from the probability distribution by using Gibbs sampling
- Look at the difference between the x(bar) and x.
- Move the weight matrix and bias vectors in a direction that minimizes this difference.

The equation is given below: -

$$W = W + lr(x^T h(x) - \tilde{x}^T h(\tilde{x}))$$
$$bh = bh + lr(h(x) - h(\tilde{x}))$$
$$bv = bv + lr(x - \tilde{x})$$

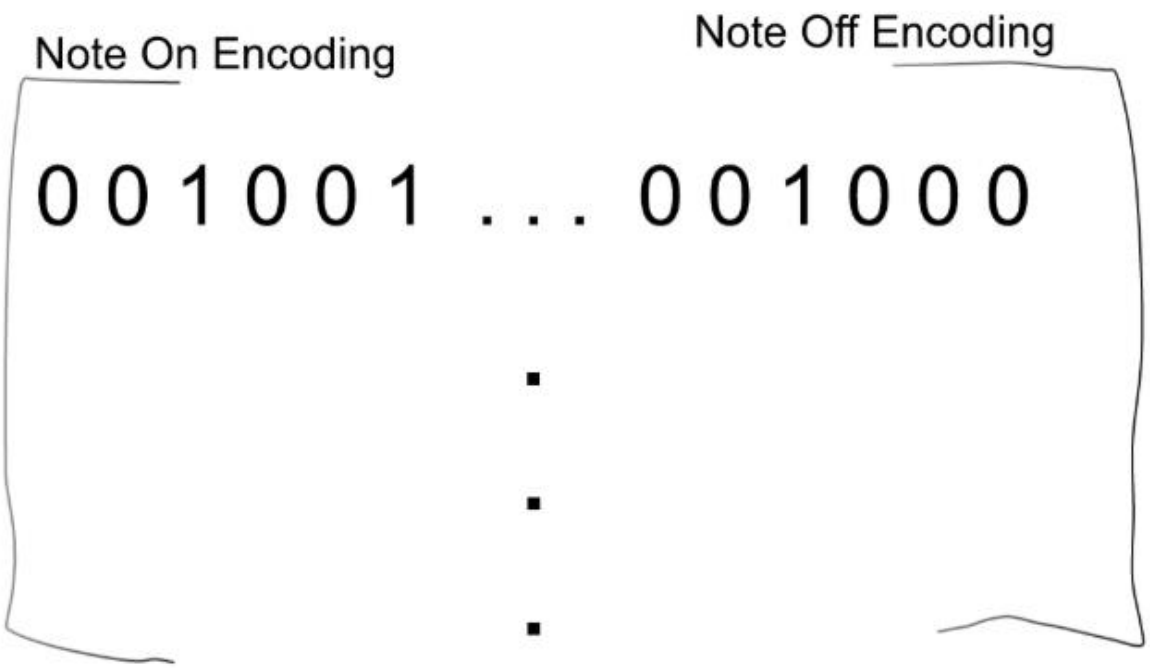
- x_t is the inital vector
- $\tilde{x}$ is the sample of x drawn from the probability distribution
- lr is the learning rate
- h is the function that takes in the values of the visible nodes and returns a sample of the hidden nodes (see step 1 of the Gibbs sampling algorithm above)

Implementation

In this project we are going to use our RBM to generate short sequences of music. Our training data will be around a hundred midi files of popular songs. Midi is a format that directly encodes musical notes - we can think of midi files as sheet music for computers. We can play midi files or convert them to mp3. Midi files encode events that a synthesizer would need to know about, such as Note-on, Note-off, Tempo changes, etc. For our purposes, we are interested in getting the following information from our midi files:

- Which note is played
- When the note is pressed
- When the note is released

We can encode each song with a binary matrix with the following structure:



The first n columns (where n is the number of notes that we want to represent) encode note-on events. The next n columns represent note-off events. So, if element M_{ij} is 1 (where $j<n$), then at timestep i, note j is played. If element $M_{i(n+j)}$ is 1, then at timestep i, note j is released.

In the above encoding, each timestep is a single training example. If we reshape the matrix to concatenate multiple rows together, then we can represent multiple timesteps in a single data vector.

Converting midi files to and from these binary matrices is relatively simple, but there are a number of annoying edge cases. All of the code for reading and writing

midi files is in the midi_manipulation.py file, which is heavily based on Daniel Johnson's midi manipulation code.